Optimization of Combinatorial Testing by Incremental SAT Solving

Akihisa Yamada, Takashi Kitamura, Cyrille Artho, Eun-Hye Choi, Yutaka Oiwa
National Institute of Advanced Industrial Science and Technology (AIST), Japan
Email: {akihisa.yamada, t.kitamura, c.artho, e.choi, y.oiwa}@aist.go.jp

Armin Biere
Institute for Formal Models and Verification, Johannes Kepler University, Austria
Email: biere@jku.at

***Abstract*—Combinatorial testing aims at reducing the cost of software and system testing by reducing the number of test cases to be executed. We propose an approach for combinatorial testing that generates a set of test cases that is as small as possible, using incremental SAT solving. We present several search-space pruning techniques that further improve our approach. Experiments show a significant improvement of our approach over other SAT-based approaches, and considerable reduction of the number of test cases over other combinatorial testing tools.**

I. Introduction

Combinatorial testing (cf. [28]) covers interactions of parameters in the system under test. A well-chosen sampling mechanism can reduce the cost of software and system testing by reducing the number of test cases to be executed. It has been empirically shown that a significant amount of defects can be detected by a test suite (a set of test cases) which covers all possible t-way interactions at least once with relatively small t [24].

In many systems, test cases involving arbitrary combinations of parameters are not necessarily executable. For instance, consider a web application that is expected to work in a various environments listed as follows:

Parameter	Values
CPU	Intel, AMD
OS	Windows, Linux, Mac
Browser	IE, Firefox, Safari

The table specifies three parameters CPU, OS and Browser, each having two or three possible values. However, not all the combinations of these values are admitted. We have the following constraints:

1) IE is available only for Windows.
2) Safari is available only for Mac.
3) Mac does not support AMD CPUs.

There is substantial work on combinatorial testing taking such constraints into account. There are *greedy* approaches [9, 12, 11, 31], approaches based on *simulated-annealing* [11, 17], and *(Max)SAT-based* approaches [23, 2, 25, 1]. They are evaluated on two measures: computation time for test case generation and the number of generated test cases.

Recent advances in greedy approaches significantly improved the speed of test case generation, even in the presence of complex constraints [31]. However, greedy approaches often require more test cases to execute, in comparison with the other two approaches. Making a test suite as small as possible can be worth a high computation time, especially when test execution is expensive as with system or field tests that require human interaction.

Simulated-annealing-based approaches often yield a fairly small test suite. However, the result is not guaranteed to be optimal; simulated annealing is non-deterministic optimization, which may fall into a local optimum far from the global one.

SAT-based approaches [23] encode the possibility of a test suite of certain size into a boolean formula, and ask a SAT solver for its solution. In theory, the approach can generate a globally optimal test suite, as well as proving its optimality. In practice, however, it is often hard to decide satisfiability when the size of allowed test suites is close to the optimum. Moreover, since it is difficult to estimate the optimum size beforehand, the SAT-based approach requires many runs of SAT solving, making the approach less scalable.

This paper aims at making optimal test case generation applicable in practice. To this end, we improve applicability of the SAT-based approach. We propose

- the use of incremental SAT solving for optimizing a combinatorial test suite,
- search-space pruning techniques that guide the effective reasoning of SAT solvers, and
- cooperation with efficient greedy testing tools.

We report on experiments using large-scale examples from [11] to verify practicality of our method. The results show that our method significantly improves the SAT-based approach, and also outperforms other approaches in terms of size of generated test suites.

The remainder of the paper is organized as follows. In the next section, we present basic notions required for the paper. Section III explains SAT encoding of combinatorial test problem. Section IV describes our test suite optimization technique, and then Section V gives a series of improvements to the proposed method. Section VI explains related work. Through experiments, Section VII measures our contribution to existing SAT-based approaches, and compares our tool with existing tools. Section VIII concludes this paper.

II. Preliminaries

A. Incremental SAT Solving

Satisfiability (SAT) [5] solvers are tools that decide if a boolean formula over a set $\mathcal{X}$ of boolean variables can be true under some *assignment* α on the boolean variables, i. e., a mapping $\alpha : \mathcal{X} \to \{\text{TRUE}, \text{FALSE}\}$. An assignment α *satisfies* a formula ϕ, written $\alpha \models \phi$, iff ϕ evaluates to TRUE after replacing every variable x in ϕ by $\alpha(x)$. Standard SAT solvers accept a boolean formula in *conjunctive normal form (CNF)*. To simplify the discussion, we assume that SAT solvers can accept arbitrary boolean formulas; cf. the well-known transformation by Tseitin [30].

Incremental SAT solving facilitates checking satisfiability for a series of closely related formulas. It is particularly important [29, 13] in the context of *bounded model checking* [4]. State-of-the-art SAT solvers like Lingeling [3] implement the assumption based incremental algorithm, as pioneered by the highly influential SAT solver MiniSAT [14].

Incremental SAT solvers remember the current state and do not just exit after checking the satisfiability of one input formula. Additional formulas can be asserted and are taken into account in further satisfiability checks. The interface of an incremental SAT solver is expressed in an object-oriented notation as follows:[1]

```
class solver {
    literal    newVar();
    void       assert(formula φ);
    bool       solve();
    assignment model; // refers to a solution if exists
};
```

B. Combinatorial Testing

We define several notions for combinatorial testing. First, we define a model of a system under test (SUT).

Definition 1. *An* SUT model *is a triple $\langle P, V, \phi\rangle$ s. t.*

- *P is a finite set whose elements are called* parameters*,*
- *$V = \{V_p\}_{p\in P}$ assigns each $p \in P$ a finite set V_p, whose elements are called the* values *of p, and*
- *ϕ, called the* SUT constraint*, is a boolean formula whose variables are pairs $\langle p, v\rangle$ of $p \in P$ and $v \in V_p$. Hereafter, we write $p.v$ instead of $\langle p, v\rangle$.*

Example 1. *Consider the web-application mentioned in the introduction. The SUT model $\langle P, V, \phi\rangle$ for this example should consist of the following parameters and values:*

$$\begin{aligned} P &= \{\text{CPU}, \text{OS}, \text{Browser}\} \\ V_{\text{CPU}} &= \{\text{Intel}, \text{AMD}\} \\ V_{\text{OS}} &= \{\text{Windows}, \text{Linux}, \text{Mac}\} \\ V_{\text{Browser}} &= \{\text{IE}, \text{Firefox}, \text{Safari}\} \end{aligned}$$

[1] Usually incremental SAT solvers have a method to specify *assumption* formulas [14], which will be valid only for the next call of satisfiability check. We do not use this functionality in this paper.

TABLE I
AN (OPTIMAL) TEST SUITE FOR THE EXAMPLE SUT.

No.	CPU	OS	Browser
1	Intel	Windows	Firefox
2	Intel	Mac	Firefox
3	Intel	Windows	IE
4	Intel	Linux	Firefox
5	Intel	Mac	Safari
6	AMD	Windows	IE
7	AMD	Linux	Firefox

The constraints 1), 2), and 3) in the introduction are expressed by the following SUT constraint:

$$\phi = \begin{cases} (\text{Browser.IE} \Rightarrow \text{OS.Windows}) & \wedge \\ (\text{Browser.Safari} \Rightarrow \text{OS.Mac}) & \wedge \\ (\text{CPU.AMD} \Rightarrow \neg\text{OS.Mac}) \end{cases}$$

A test case is a choice of values of parameters that does not violate the given constraint. It is defined as follows.

Definition 2 (test cases)**.** *A* test case *is a mapping γ from P to $\bigcup V$ which satisfies $\gamma(p) \in V_p$ for every $p \in P$ and $\widehat{\gamma} \models \phi$; here, $\widehat{\gamma}$ is the following assignment:*

$$\widehat{\gamma}(p.v) := \begin{cases} \text{TRUE} & \text{if } \gamma(p) = v \\ \text{FALSE} & \text{otherwise} \end{cases}$$

We call a set of test cases a test suite*.*

Example 2. *A test suite for the SUT model of Example 1, consisting of seven test cases, is shown in Table I.*

A basic observation supporting combinatorial testing is that faults are caused by interaction of values of a few parameters. To catch such interactions, the following notions are defined.

Definition 3 (tuples)**.** *Let an SUT $\langle P, V, \phi\rangle$ be given. A* parameter tuple *is a subset $\pi \subseteq P$ of parameters. A* value tuple*, or simply a* tuple *is a mapping τ over some $\pi \subseteq P$ s. t. $\tau(p) \in V_p$ for every $p \in \pi$.*

As in standard mathematics, we also consider a tuple τ as the following relation between parameters and values:

$$\tau = \{\, p.v \mid \tau(p) = v \text{ is defined} \,\}$$

Note that here we write $p.v$ instead of $\langle p, v\rangle$.

Definition 4 (covering tests)**.** *We say that a test case γ* covers *a tuple τ iff $\tau(p) = \gamma(p)$ whenever $\tau(p)$ is defined, and a tuple is* possible *iff it is covered by some test case (satisfying the constraints). Given a set T of tuples, we say that a test suite Γ is T-*covering *iff every $\tau \in T$ is covered by some $\gamma \in \Gamma$. The* covering test problem *is to find a T-covering test suite.*

The terms *t-way* or *t-wise* testing and *covering arrays* (cf. [28]) refer to a subclass of the covering test problems.

Definition 5 (t-way test suite)**.** *Let $\langle P, V, \phi\rangle$ be an SUT and t a positive integer. A t-*way test suite *is a T-covering test suite where T is the set of all possible tuples of size t. The value t is called the* strength *of combinatorial testing.*

TABLE II
THE ORIGINAL MATRIX FOR TABLE I AND ITS ENCODING.

No.	x_1	x_2	x_3	$x_{1=?}$ 1	$x_{1=?}$ 2	$x_{2=?}$ 1	$x_{2=?}$ 2	$x_{2=?}$ 3	$x_{3=?}$ 1	$x_{3=?}$ 2	$x_{3=?}$ 3
1	1	1	2	1	0	1	0	0	0	1	0
2	1	3	2	1	0	0	0	1	0	0	1
3	1	1	1	1	0	1	0	0	1	0	0
4	1	2	2	1	0	0	1	0	0	1	0
5	1	3	3	1	0	0	0	1	0	0	1
6	2	1	1	0	1	1	0	0	1	0	0
7	2	2	2	0	1	0	1	0	0	1	0

Example 3. *The SUT model of Example 1 has 15 possible tuples of size two. For instance,* $\{\mathrm{CPU.Intel}, \mathrm{OS.Windows}\}$, $\{\mathrm{Browser.Safari}, \mathrm{CPU.Intel}\}$*, and so on. Note that the tuple* $\{\mathrm{Browser.Safari}, \mathrm{CPU.AMD}\}$ *is not possible, although it is not explicit in the SUT constraint* ϕ*.*

The test suite in Table I is a 2-way covering test suite for the SUT model and actually it is optimal:[2]

Definition 6. *A* T*-covering test suite* Γ *is said to be* optimal *if there exists no smaller* T*-covering test suite* Γ'*, i. e.,* $|\Gamma| > |\Gamma'|$*.*

III. ENCODING COVERING TESTS INTO SAT

In this section, we present an encoding of covering test problems into SAT. We consider that P consists of k parameters and each V_p consists of g_p values (i.e., $|P| = k$ and $|V_p| = g_p$). To simplify the later discussion, we assume that parameters and values for each parameter are identified by their indices (i. e., $P = \{1, \ldots, k\}$ and $V_p = \{1, \ldots, g_p\}$).

A. Encoding Test Suites

We adopt the *original matrix model* of [22] for encoding a test suite. Let Γ be a test suite of n test cases $\gamma_1, \ldots, \gamma_n$. The *original matrix* for Γ is the following n-row k-column matrix:

$$X = \begin{pmatrix} x_1^1 & \ldots & x_k^1 \\ \vdots & & \vdots \\ x_1^n & \ldots & x_k^n \end{pmatrix}$$

Here, each i-th row of the matrix represents the test case γ_i, and each p-th column of the row represents the value of the corresponding parameter; i. e., $x_p^i = \gamma_i(p)$.

The value of x_p^i is represented using g_p boolean variables $x_{p=1}^i$, …, $x_{p=g_p}^i$, where $x_{p=v}^i$ is assigned TRUE iff $x_p^i = v$ (i. e. $\gamma_i(p) = v$). Here, x_p^i must have a unique value. Thus, we impose the following uniqueness constraint:

$$\mathsf{Unique}_n := \bigwedge_{i=1}^{n} \bigwedge_{p=1}^{k} \Big(1 = \sum_{v=1}^{g_p} x_{p=v}^i\Big)$$

The encoding is also known as *one-hot encoding* [20].

An assignment α satisfying Unique_n induces a test suite $\Gamma = \{\gamma_1, \ldots, \gamma_n\}$, which is defined as $\gamma_i(p) = v$ if and only if $\alpha(x_{p=v}^i) = \text{TRUE}$.

Example 4. *The original matrix for the test suite of Example 2 and its encoding is illustrated in Table II.*

[2] The optimality can be shown using SAT-based approaches.

B. Encoding Uniqueness Constraints

Various encodings are known for constraints of the form

$$1 = \sum_{v=1}^{g_p} x_{p=v}^i \tag{1}$$

that appear in formula Unique_n. Hnich et al. [22] adopted the *naive* encoding, that represents (1) by the following CNF:

$$\Big(\bigvee_{v=1}^{g_p} x_{p=v}^i\Big) \wedge \bigwedge_{v=1}^{g_p-1} \bigwedge_{w=v+1}^{g_p} \big(\neg x_{p=v}^i \vee \neg x_{p=w}^i\big)$$

Banbara et al. [2] adopted the *order encoding*; their encoding introduces $g_p - 1$ boolean variables $x_{p\geq 2}^i, \ldots, x_{p\geq g_p}^i$ for each row i and parameter p. The boolean variable $x_{p\geq v}^i$ represents the constraint $x_p^i \geq v$. Transitivity of the standard order $\geq$ imposes the following CNF:

$$\mathsf{Order}_n := \bigwedge_{i=1}^{n} \bigwedge_{p=1}^{k} \bigwedge_{v=1}^{g_p} \big(x_{p\geq v}^i \Leftarrow x_{p\geq v+1}^i\big)$$

Here we let $x_{p\geq 1}^i$ denote TRUE and $x_{p\geq g_p+1}^i$ denote FALSE.

The order encoding, as well as the *mixed encoding* of [2] does not introduce variables $x_{p=v}^i$ for representing the original matrix X. Our encoding uses both variables $x_{p\geq v}^i$ and $x_{p=v}^i$; the latter variables are required when considering SUT constraints. Correspondence of these variables are ensured by further imposing the following formula:

$$\mathsf{Corresp}_n := \bigwedge_{i=1}^{n} \bigwedge_{p=1}^{k} \bigwedge_{v=1}^{g_p} \big(x_{p=v}^i \Leftrightarrow x_{p\geq v}^i \wedge \neg x_{p\geq v+1}^i\big)$$

Note that Order_n and $\mathsf{Corresp}_n$ can be considered as the constraints introduced by adopting the *ladder encoding* [18] for formula (1).

C. Encoding SUT-Constraints

We adopt the straightforward translation of Nanba et al. [25] for encoding SUT-constraints. Every test case of a test suite (i. e., every row of an original matrix) must satisfy a given SUT constraint. To express this, an SUT-constraint ϕ is translated to a formula $\mathsf{Tr}_i(\phi)$ over variables $x_{p=v}^i$, by simply replacing each occurrence of $p.v$ by $x_{p=v}^i$. Using this transformation, we define the following formula:

$$\mathsf{Map}_n(\phi) := \bigwedge_{i=1}^{n} \mathsf{Tr}_i(\phi)$$

Proposition 1. *If* $\alpha \models \mathsf{Unique}_n \wedge \mathsf{Map}_n(\phi)$ *then* α *induces a test suite for the SUT model* $\langle P, V, \phi\rangle$*.*

D. Encoding Coverage Criteria

In this section we show how to encode the coverage criteria. To this end, we introduce the following notion.

Definition 7 (Cover flags)**.** *Let* T *be a set of tuples. A* T*-*cover flag array (T-CFA) *of a test suite* $\Gamma = \{\gamma_1, \ldots, \gamma_n\}$ *is an* n*-row* $|T|$*-column boolean array* $C = (c_\tau^i)$ *s. t.* γ_i *covers* τ *whenever* $c_\tau^i = \text{TRUE}$*.*

TABLE III
A 2-WAY CFA FOR THE TEST SUITE OF TABLE I.

No.	x_1	x_2	x_3	x_1x_2 11	12	13	21	22	x_2x_3 11 ... 33	x_3x_1 11 ... 31
1	1	1	2	1						
2	1	3	2			1				
3	1	1	1	1					1	1
4	1	2	2		1					
5	1	3	3			1			1	1
6	2	1	1				1		1	
7	2	2	2					1		

TABLE IV
THE ALTERNATIVE MATRIX FOR TABLE I AND ITS ENCODING.

No.	x_1	x_2	x_3	y_{12}	y_{23}	y_{31}	$y_{12=?}$ 1	2	3	4	5	$y_{23=?}$ 1 ... 5	$y_{31=?}$ 1 ... 5
1	1	1	2	1	2	3	1						
2	1	3	2	3	4	3		1					
3	1	1	1	1	1	1	1					1	1
4	1	2	2	2	3	3		1					
5	1	3	3	3	5	5			1			1	1
6	2	1	1	4	1	2				1		1	
7	2	2	2	5	3	4					1		

Example 5. *Consider again the test suite of Example 2. The CFA for the test suite is illustrated in Table III. Note that some columns, e. g. a column for '2 3' in the x_1x_2 group and one for '3 2' in the x_3x_1 group are missing in the table. These missing columns correspond to tuples that are not possible.*

We represent a T-CFA $C = (c^i_\tau)$ of a test suite Γ by boolean variables c^i_τ in an obvious manner. To ensure that C is a T-CFA, we introduce the following CNF:

$$\mathsf{CoverFlags}_n(T) := \bigwedge_{i=1}^{n} \bigwedge_{\tau \in T} \bigwedge_{p.v \in \tau} \left(c^i_\tau \Rightarrow x^i_{p=v}\right)$$

Note that $x^i_{p=v}$ holds for every $p.v \in \tau$, if and only if the tuple τ is covered by the test case γ_i. Thus, c^i_τ is true only if τ is covered by γ_i.

It is straightforward to show the following result:

Proposition 2. *If $\alpha \models \mathsf{Unique}_n \wedge \mathsf{CoverFlags}_n(T)$, then (c^i_τ) is a T-CFA of the test suite induced by α.*

Now we encode the condition that every tuple $\tau \in T$ is covered by some test case $\gamma_i \in \Gamma$. This is easily encoded as the following formula:

$$\mathsf{Covered}_n(T) := \bigwedge_{\tau \in T} \bigvee_{i=1}^{n} c^i_\tau$$

In the rest of the paper, $\mathsf{CoveringTest}_n(T, \phi)$ denotes the following formula:

$$\mathsf{Unique}_n \wedge \mathsf{Map}_n(\phi) \wedge \mathsf{CoverFlags}_n(T) \wedge \mathsf{Covered}_n(T)$$

Proposition 3. *If $\alpha \models \mathsf{CoveringTest}_n(T, \phi)$, then α induces a T-covering test suite for the SUT model $\langle P, V, \phi\rangle$.*

Proposition 4. *The formula $\mathsf{CoveringTest}_n(T, \phi)$ is satisfiable if and only if there exists a T-covering test suite Γ for the SUT model $\langle P, V, \phi\rangle$ s. t. $|\Gamma| = n$.*

E. Remark: Alternative Matrix Model and Mixed Encoding

Our encoding using cover flags looks naive but is equivalent to the SAT encoding of the *alternative matrix model* of Hnich et al. [22, 23], as well as the *mixed encoding* of Banbara et al. [2]. The *alternative matrix model* [22] represents the coverage criterion for t-way combinatorial testing. Let $\binom{P}{t}$ denote the set of all size t subsets of P. For each parameter tuple $\pi \in \binom{P}{t}$, we assume that possible tuples over domain π are indexed as $\tau^\pi_1, \ldots, \tau^\pi_{m_\pi}$.

Definition 8 ([22])**.** *The t-way* alternative matrix *for a test suite $\{\gamma_1, \ldots, \gamma_n\}$ is the n-row $\binom{k}{t}$-column matrix (y^i_π) where y^i_π is the index j s. t. $\tau^\pi_j \subseteq \gamma_i$.*

The SAT encoding of Hnich et al. and the mixed encoding of Banbara et al. represent the alternative matrix by boolean variables $y^i_{\pi=j}$ for all π and $j \in \{1, \ldots, m_\pi\}$, representing that $y^i_\pi = j$. Note here that each $y^i_{\pi=j}$ is equivalent to the cover flag c^i_τ for $\tau = \tau^\pi_j$. Hence, the formula $\mathsf{CoverFlags}_n(T)$ can be considered as a (simpler) representation of the *channeling constraints* [22].

Example 6. *Table IV presents the alternative matrix for the test suite of Example 2. In contrast to the unconstrained case, y_{12}, y_{23}, and y_{31} have only 5 values for each, since 1, 4, and 4 tuples are not possible, respectively. The right half of Table IV represents the encoding of the alternative matrix. Observe the correspondence of the encoding with the CFA of Table III.*

IV. OPTIMIZING TEST SUITES BY INCREMENTAL SAT

In this section, we introduce a test suite optimization approach using incremental SAT solving. The basic idea is inspired by the MaxSAT approach by Ansótegui et al. [1]. Thus, we first revisit their approach in the next section, and then describe our approach.

A. Optimization by Partial MaxSAT Solving

Ansótegui et al. [1] proposed use of *partial MaxSAT* [8] solvers for optimizing test suites.

Definition 9 (partial MaxSAT problems)**.** *A* soft CNF ϕ *consists of two sets of clauses: the* hard *part ϕ_h and the* soft *part ϕ_s. A* solution *to a soft CNF ϕ is an assignment that satisfies ϕ_h and maximizes the number of satisfied clauses in ϕ_s, or equivalently minimizes the number unsatisfied clauses.*

Their approach introduces n new variables $u_1, \ldots, u_n$. Each u_i encodes the property that the test case γ_i is "used" for covering some tuple $\tau \in T$. In our encoding, this condition can be expressed by assuming that the cover flags c^i_τ can be set to TRUE only if u_i is TRUE.

$$\mathsf{Using}(T) := \bigwedge_{\tau \in T} \bigwedge_{i=1}^{n} \left(c^i_\tau \Rightarrow u_i\right)$$

TABLE V
CFA AND USAGE FLAGS FOR EXAMPLE 7.

No.	u_i	x_1	x_2	x_3	x_1x_2 11	12	13	21	22	x_2x_3 11 ... 33	x_3x_1 11 ... 31
1	1	1	1	2	1						
2	1	1	3	2		1					
3	1	1	1	1	1					1	1
4	1	1	2	2		1					
5	1	1	3	3			1			1	1
6	1	2	1	1				1		1	
7	1	2	2	2					1		
8	0	1	3	2							
9	0	2	2	1							
10	0	1	3	3							

Then, the target is to minimize the number of used test cases:

$$minimize\Big(\sum_{i=1}^{n} u_i\Big) \tag{2}$$

This target is encoded as a partial MaxSAT problem using the following set of n soft-clauses:

$$\mathsf{Soft}_n := \{\neg u_1, \ldots, \neg u_n\}$$

Proposition 5. *There exists a T-covering test suite of size m w.r.t. ϕ, if and only if a solution to the soft CNF $\langle \mathsf{CoveringTest}_n(T,\phi), \mathsf{Soft}_n\rangle$ satisfies m clauses in Soft_n.*

This encoding using the new variables $u_1, \ldots, u_n$ is highly symmetric which makes efficient reasoning difficult. To break this symmetry, the following constraints are imposed:

$$\mathsf{Packed}_n := \bigwedge_{i=1}^{n-1} (u_{i+1} \Rightarrow u_i)$$

Note that Packed_n ensures that whenever m rows are used, the first m-rows are used. This symmetry breaking measure is reported to significantly improve the performance in [1].

Example 7. *Consider obtaining a test suite within 10 rows for the SUT model of Example 1. The corresponding cover-flag array and its usage flags are illustrated in Table V. The last three rows are not used to cover tuples, and thus the corresponding usage flags (u_8, u_9, and u_{10}) are falsified. The remaining seven rows constitute the test suite of Example 2.*

B. Optimization by Incremental SAT Solving

Under constraint Packed_n, the target function of (2) satisfies $\sum_{i=1}^{n} u_i < m$ if and only if u_{m+1} is assigned false. This fact suggests another way of optimization using *incremental SAT solving* [29, 13].

Suppose that the following formula is satisfiable:

$$\mathsf{CoveringTest}_n(T,\phi) \wedge \mathsf{Packed}_n$$

That is, there exists a T-covering test suite of size n. If we can further find a satisfying assignment for the following formula:

$$\mathsf{CoveringTest}_n(T,\phi) \wedge \mathsf{Packed}_n \wedge \neg u_n$$

then we know that there exists a T-covering test suite of size $n-1$. By repeating this procedure, we obtain an optimal T-covering test suite of size i when the formula

$$\mathsf{CoveringTest}_n(T,\phi) \wedge \mathsf{Packed}_n \wedge \neg u_n \wedge \cdots \wedge \neg u_{i+1} \tag{3}$$

is satisfied, and the next formula becomes unsatisfiable:

$$\mathsf{CoveringTest}_n(T,\phi) \wedge \mathsf{Packed}_n \wedge \neg u_n \wedge \cdots \wedge \neg u_{i+1} \wedge \neg u_i \tag{4}$$

Note that the only difference between (3) and (4) is the addition of the unit clause $\neg u_i$. For solving such a series of SAT instances, *incremental* SAT solving [29, 13] can be expected to improve performance. This incremental approach not only reduces the cost of generating similar formulas again and again, but also allows SAT solvers to reuse *learned clauses*, which is expected to speed-up incremental SAT solver calls and thus the overall process. Indeed, all the clauses learned for formula (3) remain valid for formula (4).

The basic procedure of the proposed approach is presented in Algorithm 1.

Algorithm 1: Optimization by Incremental SAT Solving

```
1 tool ← new solver;
2 tool.assert(CoveringTest_n(T) ∧ Packed_n);
3 for i = n, ..., 1 do
4     if tool.solve() = Unsat then
5         return Γ; // optimum found
6     Γ ← recover_test_cases(tool.model);
7     tool.assert(¬u_i);
```

V. IMPROVEMENTS

In this section, we introduce several techniques that improve the efficiency of our method.

A. Removing the New Variables

Recall that the variables $u_1, \ldots, u_n$ are added to define the target (2). Since we do not use a MaxSAT solver, we do not actually need these variables.

To get rid of these variables, we modify a CFA to a *propagating* CFA (c^i_τ), such that $c^i_\tau = \textsc{True}$ indicates that τ has been covered by some test case γ_j with $j \le i$. This is expressed by the following formula:

$$\mathsf{CoverFlags}'_n(T) := \bigwedge_{i=1}^{n} \bigwedge_{\tau\in T} \bigwedge_{p.v\in\tau} \left(c^i_\tau \Rightarrow c^{i-1}_\tau \vee x^i_{p=v}\right)$$

Here we let c^0_τ denote $\textsc{False}$.

Now, asserting $\neg u_i$ is equivalent to asserting the following:

$$\mathsf{Covered}'_i(T) := \bigwedge_{\tau\in T} c^i_\tau$$

TABLE VI
FIXING TUPLES IN A PROPAGATING CFA.

No.	x_1	x_2	x_3	x_1x_2 11	12	13	21	22	x_2x_3 11 ... 33	x_3x_1 11 ... 31
1	1	1		1	0	0	0	0		
2	1	2		1	1	0	0	0		
3	1	3		1	1	1	0	0		
4	2	1		1	1	1	1	0		
5	2	2		1	1	1	1	1		
6				1	1	1	1	1		
7				1	1	1	1	1		

B. Symmetry Breaking

Some symmetry breaking techniques [15, 19] have been successfully applied to unconstrained combinatorial testing, guiding SAT solvers' effective reasoning. In the presence of constraints, however, these techniques cannot be directly applied. Some of the basic assumptions for these methods, namely column symmetry and value symmetry, do not hold due to the existence of constraints. Thus, the previous SAT-based approach for constrained combinatorial testing by Nanba et al. [25] does not consider symmetry breaking.

Nonetheless, we still have row symmetry: Any two test cases in a test suite can be swapped without affecting the meaning of the test suite. We partially break this symmetry by fixing some values according to the set of possible tuples.[3]

Consider a parameter tuple $\pi \subseteq P$, on which possible tuples $\tau_1, \ldots, \tau_m$ exist. Every tuple τ_i must be covered by some test case, but the index of the test case can be arbitrary. Thus, we can safely impose that i-th test case covers τ_i.

Note however that value tuples can be fixed only for one parameter tuple. To fix as many values as possible, we choose a parameter tuple that consists of the maximum number of value tuples. Note also that a test suite cannot be smaller than the number of fixed value tuples. Thus, we stop optimizing when we achieved this minimum size.

Example 8. *For the SUT model of Example 1, consider fixing the parameter tuple* $\{\mathrm{CPU}, \mathrm{OS}\}$*, which is encoded as* x_1 *and* x_2 *in the original matrix. Table VI illustrates the original matrix and corresponding propagating CFA with fixing value tuples, where unfixed fields are left blank.*

C. Pruning Search Spaces

When compared to previous approaches, our approach quickly reaches a nearly optimal test suite. However, when it comes to proving the optimality, Algorithm 1 is not as good as the *iterative* approach. This phenomenon is explained as follows: In principle, all the possible assignments on variables must be considered to prove unsatisfiability of a formula. The incremental approach requires proving unsatisfiability of formula (4), which contains variables related to $x_p^i, \ldots, x_p^n$, which are irrelevant to the unsatisfiability of the formula. These variables are not present in $\mathsf{CoveringTest}_{i-1}(T, \phi)$, which would be checked in the *iterative* approach.

[3] An essentially equivalent idea is mentioned as *preprocessing* [32, p.65].

In the incremental approach, we can virtually prune the search space for these irrelevant variables, by fixing the assignment of them by asserting *unit clauses* (i. e., clauses containing only one literal) depending on their current assignment. In principle, the incremental approach can be optimized if the back-end SAT solver provides a dedicated function to remove variables, e. g. methods 'release' of MiniSAT and 'melt' of Lingeling.

The overall procedure is illustrated in Algorithm 2.

Algorithm 2: Improved Test Suite Optimization

```
1  tool.assert(Unique_n ∧ Map_n(φ) ∧ CoverFlags'_n);
2  Fix m value tuples according to Section V-B;
3  for i = n, ..., m do
4      if tool.solve() = Unsat then
5          return Γ; // optimum found
6      Γ ← recover_test_cases(tool.model);
7      tool.assert(Covered'_i(T));
8      for p ∈ P, v ∈ V_p do // pruning
9          if tool.model[x^i_{p=v}] = TRUE then
10             tool.assert(x^i_{p=v});
11         else
12             tool.assert(¬x^i_{p=v});
13 return Γ ; // achieved the minimum size
```

D. Cooperation with Greedy Covering Test Tools

There are two main challenges in applying the SAT-based test suite optimization techniques: finding a reasonably small upper bound for the size of test suites, and enumerating all the possible tuples. For the latter problem, a SAT-based approach has been proposed [25].

We solve both of the problems using existing test case generation tools, such as PICT [12] or ACTS [7]. These tools do not aim at generating an optimal covering test suite. Instead, they aim at quickly generating a covering test suite of reasonable size. Thus, the size of a test suite generated by such tools is a good starting point for optimization. Moreover, the set of possible tuples can be obtained by enumerating value tuples that appear in the test suite.

VI. RELATED WORK

A. Test-Suite Minimization

The *minimization* problem of covering arrays was proposed by Hartman and Raskin [21], and generalized by Blue et al. [6] for test suites with (implicit) SUT constraints. Their minimization problems are different from our *optimization* problem; given an initial test suite Γ and a coverage criterion (e. g. 2-way), the minimization problem of [21] is to find a smallest subset $\Gamma' \subseteq \Gamma$ that satisfies the given coverage criterion. Our optimization problem does not impose any relation between Γ and Γ', thus can obtain smaller test suites in general. On the other hand, our approach requires constraints be explicit.

TABLE VII
COMPARISON WITH EXISTING SAT-BASED APPROACHES FOR 2-WAY COVERAGE.

			ACTS		iterative		MaxSAT		Algorithm 1		Algorithm 2	
Name	Model	Constraints	size	time	size	time	size	time	size	time	size	time
SPIN-S	$2^{13}4^5$	2^{13}	26	0.92	19*	2.60	19*	19.39	19*	2.84	19*	**1.70**
SPIN-V	$2^{42}3^24^{11}$	$2^{47}3^2$	45	1.46	31	264.50	35	1186.64	31	606.46	31	**87.46**
GCC	$2^{189}3^{10}$	$2^{37}3^3$	23	1.65	15	687.78	18	2683.66	15	855.66	15	**135.33**
Apache	$2^{158}3^84^45^16^1$	$2^33^14^25^1$	33	1.10	30*	451.18	30*	2559.07	30*	760.67	30*	**144.95**
Bugzilla	$2^{49}3^14^2$	2^43^1	19	0.55	16*	5.24	16*	15.44	16*	5.28	16*	**2.79**
1	$2^{86}3^34^15^56^2$	$2^{20}3^34^1$	48	4.04	38	983.76	42	1892.65	38	620.18	**37**	809.15
2	$2^{86}3^34^35^16^1$	$2^{19}3^3$	32	2.07	30*	41.10	30*	261.36	30*	42.31	30*	**24.96**
3	$2^{27}4^2$	2^93^1	19	0.73	18*	1.04	18*	2.84	18*	1.07	18*	**1.00**
4	$2^{51}3^44^25^1$	$2^{15}3^2$	22	1.22	20*	6.59	20*	31.02	20*	15.81	20*	**4.50**
5	$2^{155}3^74^35^56^4$	$2^{32}3^64^1$	54	5.33	46	3194.67	53	1520.93	46	3437.61	**45**	2344.78
6	$2^{73}4^36^1$	$2^{26}3^4$	25	1.83	24*	9.38	24*	49.22	24*	26.23	24*	**8.92**
7	$2^{29}3^1$	$2^{13}3^2$	12	0.68	9	1.11	9	9.75	9	1.15	9	**0.77**
8	$2^{109}3^24^25^36^3$	$2^{32}3^44^1$	47	4.10	37	1277.83	43	1967.39	**36***	1294.33	**36***	**501.99**
9	$2^{57}3^14^15^16^1$	$2^{30}3^7$	22	1.87	20*	5.91	20*	10.20	20*	4.93	20*	**3.85**
10	$2^{130}3^64^55^26^4$	$2^{40}3^7$	47	6.67	41	2069.71	45	1288.32	**39**	3078.28	**39**	**2195.97**
11	$2^{84}3^44^25^26^4$	$2^{28}3^4$	47	431	39	2766.41	44	2027.18	39	969.36	39	**575.84**
12	$2^{136}3^44^34^16^3$	$2^{23}3^4$	43	4.73	36*	666.02	38	3223.91	36*	642.72	36*	**127.09**
13	$2^{124}3^44^15^26^2$	$2^{22}3^4$	40	2.58	36*	269.53	36*	1409.24	36*	346.79	36*	**79.60**
14	$2^{81}3^54^36^3$	$2^{13}3^2$	39	1.54	36*	79.07	36*	409.35	36*	124.75	36*	**30.95**
15	$2^{50}3^44^15^26^1$	$2^{20}3^2$	32	1.40	30*	10.14	30*	62.82	30*	25.84	30*	**5.73**
16	$2^{81}3^34^26^1$	$2^{30}3^4$	25	2.04	24*	10.37	24*	68.44	24*	37.61	24*	**10.11**
17	$2^{128}3^34^25^16^3$	$2^{25}3^4$	41	3.57	36*	333.16	36*	2531.96	36*	394.04	36*	**87.71**
18	$2^{127}3^23^34^66^2$	$2^{23}3^44^1$	52	3.18	40	2982.15	48	2599.03	**39**	3419.77	40	926.92
19	$2^{172}3^94^95^36^4$	$2^{38}3^5$	51	4.88	44	2950.88	50	2916.59	43	2919.63	**42**	1572.37
20	$2^{138}3^44^55^46^7$	$2^{42}3^6$	60	8.06	55	2591.95	(60)	(10.28)	58	3353.96	**54**	1382.47
21	$2^{76}3^34^25^16^3$	$2^{40}3^6$	39	2.63	36*	53.96	36*	172.24	36*	70.89	36*	**21.75**
22	$2^{72}3^44^16^2$	$2^{31}3^4$	37	1.77	36*	12.84	36*	75.86	36*	31.79	36*	**12.70**
23	$2^{25}3^16^1$	$2^{13}3^2$	14	0.81	12*	1.02	12*	2.13	12*	0.98	12*	**0.88**
24	$2^{110}3^25^36^4$	$2^{25}3^4$	48	3.45	42	2386.82	46	1998.46	42	1452.85	**41**	1917.11
25	$2^{118}3^64^25^26^6$	$2^{23}3^34^1$	52	3.92	49	1570.63	(52)	(3.59)	49	3253.74	**48**	2748.84
26	$2^{87}3^14^35^4$	$2^{28}3^4$	34	2.87	27	351.42	28	1454.10	**26**	**862.92**	**26**	1337.59
27	$2^{55}3^24^25^16^2$	$2^{17}3^3$	37	1.58	36*	9.50	36*	69.03	36*	31.69	36*	**9.19**
28	$2^{167}3^{16}4^25^36^6$	$2^{31}3^6$	57	4.61	52	2870.53	(57)	(5.34)	50	3547.34	**49**	2269.73
29	$2^{134}3^75^3$	$2^{19}3^3$	29	1.81	25*	244.19	25*	1218.91	25*	352.88	25*	**63.10**
30	$2^{73}3^34^3$	$2^{20}3^2$	22	1.96	16*	31.41	16*	116.89	16*	34.15	16*	**8.65**

B. Covering Array Optimization

Nayeri et al. [26, 27] proposed a randomized optimization of covering arrays and reported significant improvements over non-optimal algorithms, especially on relatively large problems (requiring hundreds of rows). Their approach exploits "don't care" fields, the fields that can have arbitrary value without affecting the coverage.

Their approach cannot be easily extended for optimization of test suites in the presence of constraints, since a parameter involved in constraints can rarely be a "don't care". To check if a field is a "don't care", one must check if all candidate values of the field do not violate the given constraints. Also, their approach does not guarantee optimality.

On the other hand, their approach is more scalable than ours. We leave it for future work to apply heuristic optimization under the presence of constraints.

C. Simulated Annealing

Simulated annealing has been adapted for constructing covering arrays [10], and extended for constraints [16, 17]. While these approaches use a very different technique to ours, we have a similarity: When they find a covering test suite, they do not immediately output the test suite but try to get a smaller one. Thus, CASA,[4] a combinatorial testing tool based on simulated annealing, is compared in the experiments.

D. Optimization by Iterative SAT Solving

The original approach of Hnich et al. [22] requires calling SAT solvers *iteratively*. The basic idea of the approach is to find an optimal test suite by applying Proposition 4 until the encoded formula becomes unsatisfiable, decreasing the size of the test suite. Whenever the size is decreased, a back-end SAT solver is initialized for checking satisfiability of the new formula. Banbara et al. [2] basically succeeded this approach, and Nanba et al. [25] applied binary search.

VII. EXPERIMENTS

We have implemented our incremental SAT-based approach in a tool called Calot. For comparison, we also implemented other SAT-based approaches: iterative SAT-based and MaxSAT-based. As a back-end SAT solver, we choose Lingeling[5] for both iterative and incremental approaches. For a MaxSAT solver, we use SAT4J.[6]

[4] Available at http://cse.unl.edu/~citportal/.

[5] Version ayv, available at http://fmv.jku.at/lingeling/, see also [3].

[6] Version 2.3.5.v20130525, available at http://www.sat4j.org/.

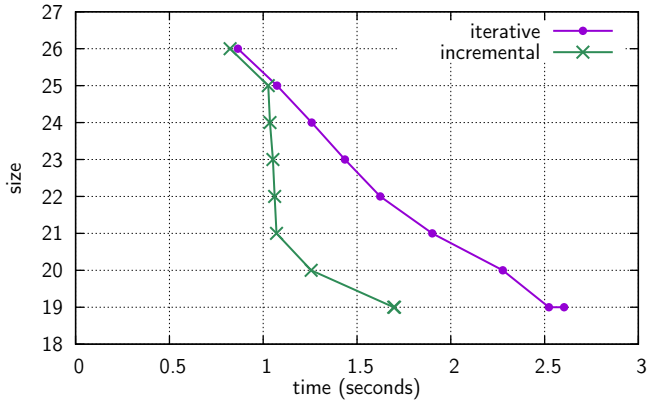

Fig. 1. Speed of optimization: SPIN-S

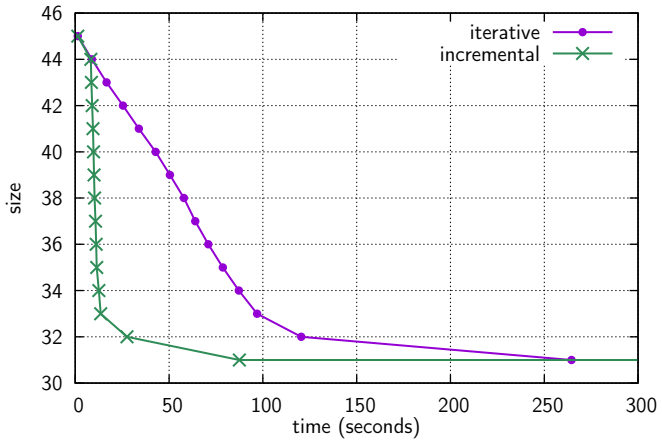

Fig. 2. Speed of optimization: SPIN-V

Experiments were conducted on the 35 benchmark problems designed by Cohen et al. [11], performed on a PC with a Quad-Core Intel Xeon E5 3.7GHz and 64GB Memory running on Mac OS 10.9.4. For each run of test case generation, the timeout is set to 3600 seconds.

A. Comparison with Existing SAT-Based Approaches

First, we compare our optimization method using incremental SAT solving and the existing methods using iterative SAT solving and MaxSAT solving. To clarify the effect of incremental SAT solving, we apply the symmetry breaking of Section V-B equally for all these methods.

The results are shown in Table VII. In the table, the size of an SUT model is expressed as $g_1^{k_1} g_2^{k_2} ... g_n^{k_n}$, which means that for each i there are k_i parameters that have g_i values. The complexity of an SUT-constraint is expressed in the same format, but this time it means that there are k_i constraints that involve g_i parameters.

The 'iterative' column indicates the approach of Section VI-D, 'MaxSAT' indicates the MaxSAT based approach of Section IV-A, 'Algorithm 1' indicates the basic incremental approach (with symmetry breaking), and 'Algorithm 2' indicates the improved version with all the improvements of Section V. The sizes of test suites obtained by each method are listed

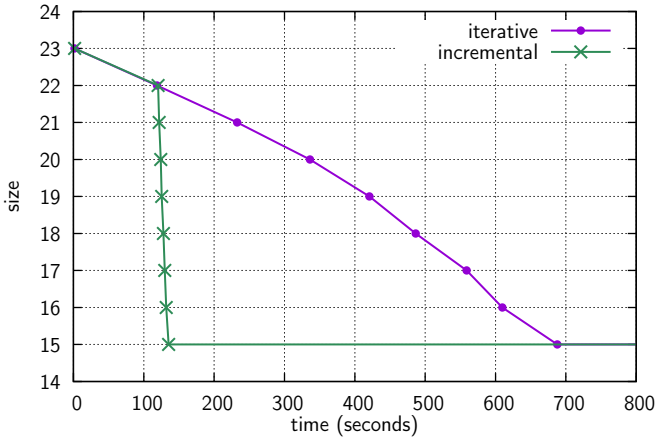

Fig. 3. Speed of optimization: GCC

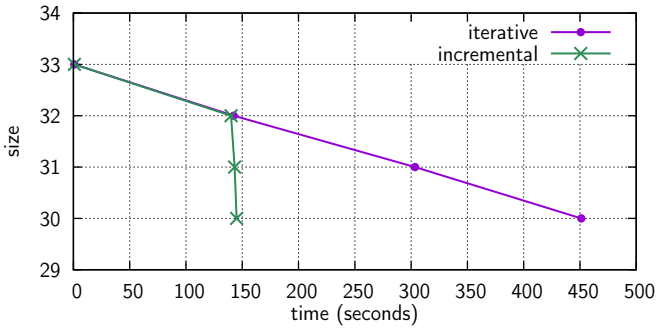

Fig. 4. Speed of optimization: Apache

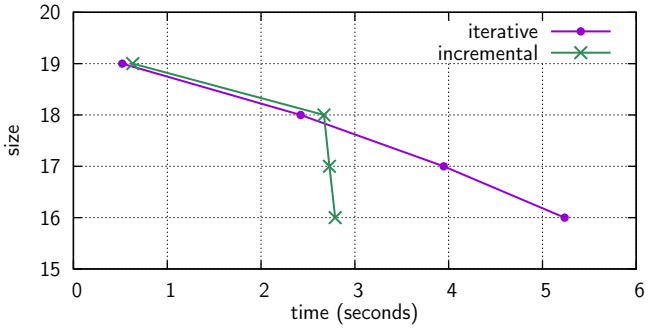

Fig. 5. Speed of optimization: Bugzilla

in the 'size' column, where the '*' marks indicate that the optimality could be proven. The 'time' column indicates the time to achieving the size or proving optimality.

As described in Section V-D, we employ a greedy testing tool to obtain the initial size of test suites and the list of all possible tuples. In our experiments, we use ACTS for this purpose. Thus the table also shows the results for ACTS. The 'time' columns for all methods include the time consumed by ACTS. Parentheses indicate that the test suite could not be improved from the result of ACTS, although it was the case only for the MaxSAT-based approach on three benchmarks.

Highlighted sizes indicate that the size could be achieved only by one method. If several methods obtained the best size, best time to achieving the size is highlighted.

For all the benchmarks, our incremental approach (Algorithms 1 and 2) achieved the smallest size among others. In case the existing methods also achieved the same size, Algorithm 2 gets to the solution much faster than the others in many cases. For a few benchmarks, namely No. 3, 6, 16, and 22, the improvement is unclear; in these exceptions ACTS returned test suites which are close to the optimum. In such cases, the incremental approach gains little since the number of SAT solving is limited.

We observe that Algorithm 1 improves over the MaxSAT approach, but it does not have a decisive advantage over the iterative approach. Thanks to the improvements proposed in Section V (except for symmetry breaking), Algorithm 2 became the clear winner.

B. Speed of Optimization

In this section, we clarify the improvements in the speed of optimization that is obtained by the incremental SAT solving. For this purpose, we look in detail at the first five benchmarks (SPIN-S, SPIN-V, GCC, Apache, and Bugzilla), which have been derived from real-world examples by Cohen et al. [11].

Figures 1 to 5 illustrate the relation between the runtime and the size of test suites obtained so far for the iterative approach and the incremental approach (Algorithm 2).

The left-end of each curve indicates the initial size of test suites returned by ACTS and its computation time. Observe that the first segments of the two curves almost overlap in every graph. This is because the initial problems passed to a SAT solver are the same for iterative and incremental approaches. From the second segment, the incremental approach performs a steep decrease of the size until it slows down by reaching the optimum. On the other hand, we do not observe a notable speed-up in the iterative approach. The horizontal right-most segments in the figures for the benchmarks SPIN-S, SPIN-V, and GCC indicate the time consumed to prove optimality (although it could not be ensured in an hour for SPIN-V and GCC). For the benchmarks Apache and Bugzilla, our tool proved optimality immediately due to the discussion of Section V-B.

C. Comparison with CASA

Table VIII compares our tool with the simulated-annealing-based tool CASA. Both tools were run with timeout of an hour, although CASA did not hit the timeout.

We measure the time Calot achieved the size of CASA in the column 'time (#CASA)'. Values in parenthesis indicates the time for ACTS; for these benchmarks, ACTS generated fewer test cases than CASA did. The column 'time (best)' indicates the time Calot achieved the best size or ensured the optimality.

Let us first compare CASA and Calot in terms of the size of test suites. For 21 out of the 35 benchmarks, our tool eventually yielded a smaller test suite than CASA. For other 13 benchmarks, these tools yielded test suites of the same size. There is only one exception (benchmark No. 20) for which Calot did not reach the size CASA did within the time limit.

TABLE VIII
COMPARISON WITH CASA FOR 2-WAY COVERAGE.

	CASA		Calot (Algorithm 2)		
Name	size	time	best size	time (#CASA)	time (best)
SPIN-S	**19**	2.70	**19***	**1.70**	**1.70**
SPIN-V	38	19.18	**31**	**10.30**	87.46
GCC	21	1317.08	**15**	**121.83**	**135.33**
Apache	**30**	**115.34**	**30***	144.95	144.95
Bugzilla	**16**	**2.47**	**16***	2.79	2.79
1	49	47.09	**37**	**(2.98)**	809.15
2	**30**	**15.06**	**30***	24.96	24.96
3	**18**	**0.63**	**18***	1.00	1.00
4	21	5.17	**20***	**4.39**	**4.50**
5	54	69.19	**45**	**(5.79)**	2344.78
6	**24**	8.99	**24***	**8.92**	**8.92**
7	**9**	**0.55**	**9**	0.77	0.77
8	43	**53.73**	**36***	93.97	501.99
9	21	5.49	**20***	**3.76**	**3.85**
10	46	**144.20**	**39**	153.19	2195.97
11	42	868.16	**39**	**67.58**	575.84
12	42	**110.32**	**36***	112.99	127.09
13	**36**	107.91	**36***	**79.60**	**79.60**
14	41	13.55	**36***	**(1.42)**	30.95
15	**30**	12.32	**30***	**5.73**	**5.73**
16	**24**	12.24	**24***	**10.11**	**10.11**
17	39	**26.73**	**36***	81.36	87.71
18	47	**36.02**	**40**	178.74	926.92
19	55	93.75	**42**	**(4.19)**	1572.37
20	**53**	**1689.58**	54	-	1382.47
21	**36**	141.88	**36***	**21.75**	**21.75**
22	**36**	**8.33**	**36***	12.70	12.70
23	17	1.22	**12***	**(0.72)**	0.88
24	46	**68.11**	**41**	81.42	1917.11
25	49	**959.15**	**48**	1808.68	2748.84
26	32	**16.19**	**26**	20.55	1337.59
27	**36**	**6.02**	**36***	9.19	9.19
28	55	**175.71**	**49**	495.10	2269.73
29	30	70.42	**25***	**(1.59)**	63.10
30	21	**3.85**	**16***	7.77	8.65

In terms of the speed of optimization, we cannot observe a clear tendency here. For some benchmarks (e. g. GCC, No. 11 and 21) Calot achieved the size of CASA more quickly, but for some others (e. g. No. 20 and 25) CASA outperformed Calot. We leave it for future work to analyze the reason.

VIII. CONCLUSION

In this paper, we proposed the use of incremental SAT solving for optimizing combinatorial testing. We presented a SAT encoding of combinatorial testing, and related it with the existing encoding by Hnich et al. [23]. Then we introduced an algorithm for optimizing test suites using incremental SAT solving, and proposed a series of improvements of search-space pruning and cooperation with efficient greedy combinatorial testing tools. We compared our method and other approaches through experiments. The experimental results confirmed the significant improvement of computation time by our work against other (Max)SAT-based approaches. Further, our method outperformed other approaches in terms of the number of test cases, and its speed was competitive to the simulated-annealing-based tool CASA.

For future work, we consider further improving performance. When $t = 3$, our preliminary experiments suggested that one hour timeout might not be sufficient for large

scale benchmarks. For instance, benchmark Apache induces 8 millions of possible 3-way tuples. How to handle such an explosion of possible tuples will be our next challenge.

Acknowledgement

This work is partly supported by JST A-STEP grant AS2524001H.

References

[1] C. Ansótegui, I. Izquierdo, F. Manyà, and J. Torres-Jiménez. A max-SAT-based approach to constructing optimal covering arrays. In *CCIA 2013*, volume 256 of *FAIA*, pages 51–59, 2013.

[2] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue. Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. In *LPAR-17*, volume 6397 of *LNCS*, pages 112–126, 2010.

[3] A. Biere. Yet another local search solver and Lingeling and friends entering the SAT Competition 2014. In *SAT Competition 2014*, volume B-2014-2 of *Department of Computer Science Series of Publications B*, pages 39–40. University of Helsinki, 2014.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS 1999*, volume 1579 of *LNCS*, pages 193–207, 1999.

[5] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *FAIA*. IOS Press, February 2009.

[6] D. Blue, I. Segall, R. Tzoref-Brill, and A. Zlotnick. Interaction-based test-suite minimization. In *ICSE 2013*, pages 182–191, 2013.

[7] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn. Combinatorial testing of ACTS: A case study. In *ICST 2012*, pages 591–600, 2012.

[8] B. Cha, K. Iwama, Y. Kambayashi, and S. Miyazaki. Local search algorithms for partial MAXSAT. In *AAAI/IAAI*, pages 263–268, 1997.

[9] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans. Software Eng.*, 23 (7):437–444, 1997.

[10] M. B. Cohen, C. J. Colbourn, and A. C. Ling. Augmenting simulated annealing to build interaction test suites. In *ISSRE 2003*, pages 394–405, 2003.

[11] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Trans. Software Eng.*, 34(5):633–650, 2008.

[12] J. Czerwonka. Pairwise testing in real world. In *PNSQC 2006*, pages 419–430, 2006.

[13] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[14] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT 2003*, volume 2919 of *LNCS*, pages 502–518, 2004.

[15] P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *CP 2002*, volume 2470 of *LNCS*, pages 462–477, 2002.

[16] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *SSBSE 2009*, pages 13–22, 2009.

[17] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empir. Software Eng.*, 16:61–102, 2011.

[18] I. P. Gent and P. Nightingale. A new encoding of alldifferent into SAT. In *International Workshop on Modelling and Reformulating Constraint Satisfaction*, pages 95–110, 2004.

[19] A. Grayland, I. Miguel, and C. M. Roney-Dougal. Snake lex: An alternative to double lex. In *CP 2009*, volume 5732 of *LNCS*, pages 391–399, 2009.

[20] D. M. Harris and S. L. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann, Burlington, 2007. ISBN 978-0-12-370497-9.

[21] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *Discrete Math.*, 284:149–156, 2004.

[22] B. Hnich, S. Prestwich, and E. Selensky. Constraint-based approaches to the covering test problem. In *CSCLP 2004*, volume 3419 of *LNAI*, pages 172–186, 2005.

[23] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3):199–219, 2006.

[24] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng.*, 30(6):418–421, 2004.

[25] T. Nanba, T. Tsuchiya, and T. Kikuno. Using satisfiability solving for pairwise testing in the presence of constraints. *IEICE Trans. Fundamentals*, E95-A(9), 2012.

[26] P. Nayeri, C. J. Colbourn, and G. Konjevod. Randomized postoptimization of covering arrays. In *IWOCA 2009*, volume 5874 of *LNCS*, pages 408–419, 2009.

[27] P. Nayeri, C. J. Colbourn, and G. Konjevod. Randomized postoptimization of covering arrays. *Eur. J. Combin.*, 34: 91–103, 2013.

[28] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11, 2011.

[29] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *CHARME 2001*, volume 2144 of *LNCS*, pages 58–70, 2001.

[30] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 2(115-125):10–13, 1968.

[31] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker, and D. R. Kuhn. An efficient algorithm for constraint handling in combinatorial test generation. In *ICST 2013*, pages 242–251. IEEE, 2013.

[32] J. Zhang, Z. Zhang, and F. Ma. *Automatic Generation of Combinatorial Test Data*. Springer Briefs in Computer Science. 2014.